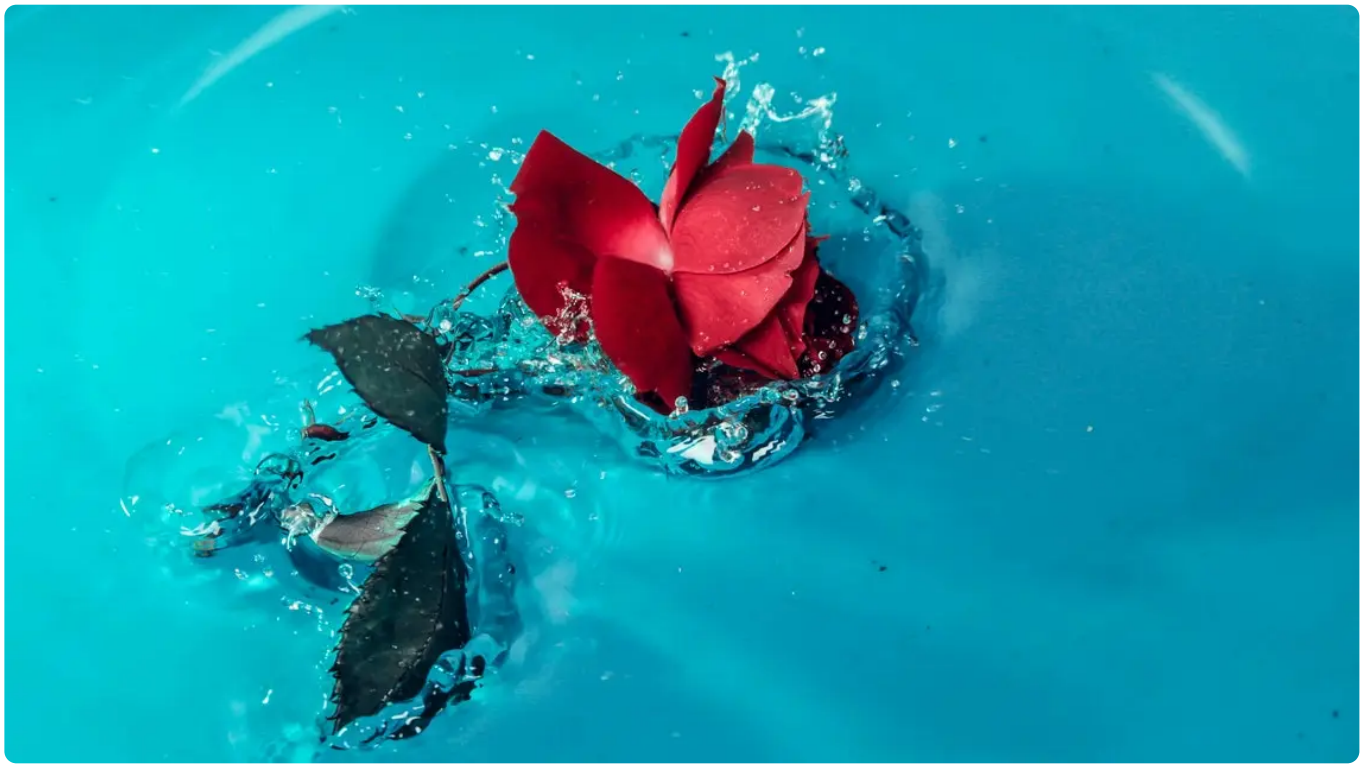
31 | 容量场景：决定线上容量场景的关键因素是什么？

2021-12-28 高楼

《全链路压测实战30讲》 课程介绍 >



讲述：高楼

时长 19:47 大小 18.12M

你好，我是高楼。

上节课，我们对场景进行了一个预压测，提前解决了一些基础性的问题。这节课呢，我们就可以直接上大的压力来执行容量场景了。容量场景的目的就是验证线上环境能不能满足需求，最后给我们一个明确的答案。

通常在真实的项目中，我们都会有总 TPS 的需求，也会有详细的 TPS 需求（每个接口应该在多长的响应时间内达到多少 TPS）。在第 3 讲的压测方案中，我们已经有了一个指标表，我也把它直接放在下面了。

序号	操作	接口	目标 TPS	TPS 标准方差	响应时间	响应时间标准方差
1	打开首页	/home/content	330	5%	200ms	5%
2	用户登陆	/sso/login(这个接口还会调用/oauth/token、/sso/loadByUsername)	150	5%	200ms	5%
3	用户信息查询	/member/address/list	150	5%	200ms	5%
4	商品加入购物车	/cart/add	150	5%	200ms	5%
5	查询购物车信息	/cart/list	90	5%	200ms	5%
6	购物车信息确定订单	/order/generateConfirmOrder	90	5%	200ms	5%
7	生成订单信息	/order/generateOrder	90	5%	200ms	5%
8	支付前查询订单列表	/order/list	90	5%	200ms	5%
9	支付订单信息	/order/paySuccess	90	5%	200ms	5%
10	支付后查询订单详情	/order/detail/${orderId}	150	5%	200ms	5%

极客时间

这里的目标 TPS 都加在一起是 1350，后面响应时间也给出了个大概的值（200ms 以下），这也算是个比较明确的指标了。下面我们就通过容量场景来看看这个目标能不能达到，要是能达到，我们就可以给出“测试通过”的结论了。

下面我们先运行一下容量场景，这里用 JMeter 或 GoReplay 来运行都是可以的。

不过在这之前我们得知道，虽然在前面的预压测中，我们有过把 TPS 调到 900 左右的时候。但是现在环境也经过了好几轮的折腾，各应用也重启过好几次了，我们不知道各个服务是不是还是在预压测的那个状态。

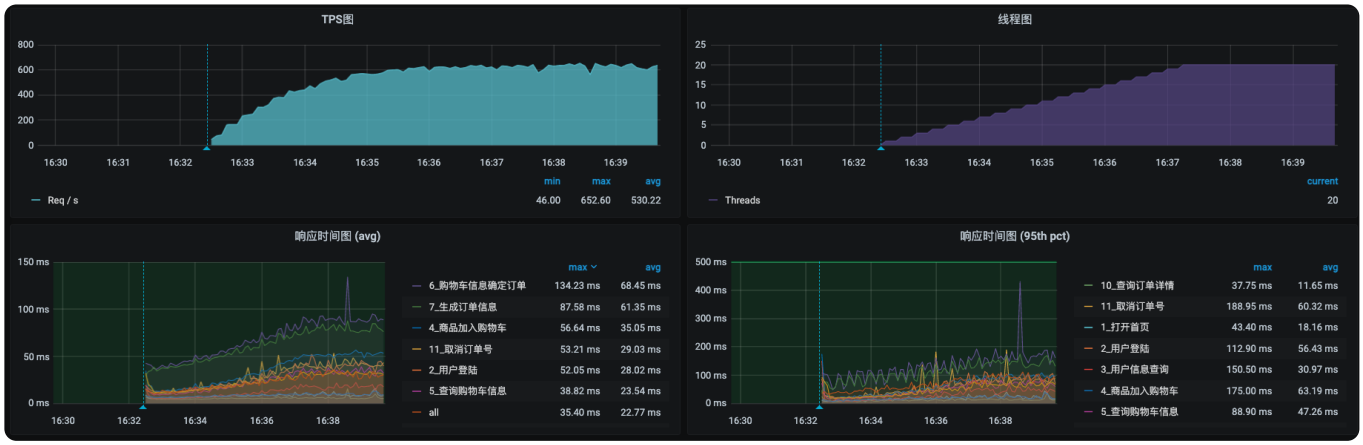
这也是我们在性能项目中经常会面对的问题：有些指标在之前执行的场景中一切良好，但在解决其他问题的过程中，突然就出现了性能变差的情况。这种情况经常会随机地出现，所以如果有性能工程师把某一次压力良好的数据直接当成最终的性能结果，就这样把它写进性能报告里，这肯定是应付差事的表现。

果然，在我们的这个容量场景中就出现了性能衰减的问题。我们一起来看看数据。

第一阶段

压力场景数据

先来看压力场景的数据：



在预压测中，同样的 20 个压力线程，我们达到过 900 左右的 TPS。但是现在只有 600 左右了。这里要说明一下，在性能项目中，由于环境的变化、版本的变化、数据量级的变化等，TPS 有下降也是时有发生的事情，遇到这样的事情，首先是不要慌，先分析下是什么原因，再来看如何解决。

从响应时间上看，有几个事务的响应时间已经到 80ms 左右了。这和之前的数据相比差了很多。

我们还是按照性能分析七步法来分析一下问题在哪。

因为架构图我在前面也提了很多次了，相信你跟着我这个专栏走下来，这个架构图是可以记得住的。所以在这里我就不再强调了。

在性能分析七步法中，“拆分时间”和“全局监控”这两步我经常是不分先后的，因为这两步都有可能直接看到性能瓶颈的问题方向。如果我们分析全局监控看不到问题的方向，就可以回去拆分响应时间来判断瓶颈点在哪了。对于性能来说，核心其实就是响应时间的快慢，我们所有的努力都是为了让响应时间更快。当响应时间快了之后，单节点的容量也就高了。然后我们可以再进行扩展性测试，来判断需要多少资源才能支撑整个架构的容量。

下面我们先来看看全局监控的计数器。

全局监控分析

前面我们讲过全局监控分析的设计和落地实现，我们按照这个思路查看一下全局监控计数器。在微服务分布式架构中，全局监控主要的视角包括：Node 级（系统级）、微服务级和 Pod 级。从 Node 级来看：

All：节点信息明细

节点	状态	Pod上	Pod	CPU总	CPU使用	CPU使用%	CPU请	CPU请求%	CPU限	CPU限	内存总	内存使用	内存使用	内存请求	内存请求	内存限制	内存限	磁盘总	磁盘使用	磁盘使用
s6	Rea...	110	9	2	1.61	80.5%	0.700	35%	1.10	55.0%	7.54 GiB	4.03 GiB	53.5%	662.00 ...	8.57%	3.00 GiB	39.8%	52.91 GiB	33.99 GiB	64.2%
s12	Rea...	110	7	8	3.58	44.7%	2.21	27.6%	3.11	38.9%	14.91 GiB	9.65 GiB	64.7%	4.66 GiB	31.2%	7.01 GiB	47.0%	46.75 GiB	30.65 GiB	65.6%
s11	Rea...	110	17	8	1.95	24.4%	4.60	57.5%	5.40	67.5%	14.91 GiB	7.90 GiB	53.0%	8.15 GiB	54.7%	10.08 GiB	67.6%	46.75 GiB	26.45 GiB	56.6%
s10	Rea...	110	17	8	1.11	13.8%	6.00	75.0%	8.30	104%	14.91 GiB	5.99 GiB	40.2%	8.45 GiB	56.7%	10.97 GiB	73.6%	46.75 GiB	15.32 GiB	32.8%
s7	Rea...	110	9	2	0.221	11.0%	0.800	40%	3.10	155%	7.54 GiB	3.96 GiB	52.6%	1.65 GiB	21.8%	5.00 GiB	66.3%	52.91 GiB	35.60 GiB	67.3%
s8	Rea...	110	15	8	0.785	9.81%	4.56	57.1%	12.4	155%	14.91 GiB	8.79 GiB	58.9%	7.30 GiB	48.9%	14.67 GiB	98.5%	46.75 GiB	20.81 GiB	44.5%
s5	Rea...	110	10	2	0.114	5.70%	0.950	47.5%	0.100	5%	7.54 GiB	1.34 GiB	17.7%	290.00 ...	3.76%	2.33 GiB	31.0%	52.91 GiB	23.80 GiB	45.0%
s9	Rea...	110	21	8	0.135	1.57%	5.72	71.5%	18.6	233%	14.91 GiB	8.73 GiB	58.6%	9.79 GiB	65.7%	18.00 GiB	121%	46.75 GiB	12.04 GiB	25.8%

s6 CPU 资源占用比较高。它的算法是用 CPU 使用核数 /CPU 总核数得到的数据，所以可能会让你感觉这个值不精准。比如上面的 s6 节点的 CPU 使用率是 80.5%，这个数据就跟主机监控视图里的数据不一样：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	1.60 week	15.00 GiB	8	0.53	11.01%	43.20%	40.94%	66.39 kB/s	101.78 kB/s	10	124	80.56 Mb/s	97.22 Mb/s
s11	s11	1.60 week	15.00 GiB	8	2.52	39.21%	57.28%	71.89%	675.25 kB/s	12.33 MB/s	17	161	502.34 Mb/s	126.49 Mb/s
s12	s12	4.68 day	15.00 GiB	8	5.54	52.60%	47.97%	83.39%	10.45 kB/s	5.84 MB/s	10	176	109.82 Mb/s	357.32 Mb/s
s5	s5	2.15 week	7.64 GiB	2	0.69	19.75%	32.06%	50.78%	0 B/s	99.44 kB/s	298	153	13.72 Mb/s	4.58 Mb/s
s6	s6	2.15 week	7.64 GiB	2	13.09	92.72%	32.07%	74.27%	11.23 kB/s	7.18 MB/s	22	440	28.82 Mb/s	26.37 Mb/s
s7	s7	2.15 week	7.64 GiB	2	0.48	18.69%	77.05%	75.98%	0 B/s	722.65 kB/s	10	238	3.91 Mb/s	6.67 Mb/s
s8	s8	1.60 week	15.00 GiB	8	1.08	17.13%	53.19%	57.11%	0 B/s	5.02 MB/s	10	407	20.33 Mb/s	18.54 Mb/s
s9	s9	2.15 week	15.00 GiB	8	0.79	14.37%	63.16%	32.25%	0 B/s	49.46 kB/s	11	338	105.60 Mb/s	127.11 Mb/s

在主机监控视图里，我们看到 s6 的 CPU 使用已经超过 90% 了。这个值有点太高了，应该从它开始往下分析。

定向监控分析

我们再来看下定向监控。登录 s6 主机，查看资源。

复制代码

```
top - 16:44:01 up 15 days,  1:10,  1 user,  load average: 16.38, 13.87, 7.53
Tasks: 127 total,   2 running, 125 sleeping,   0 stopped,   0 zombie
%Cpu0  : 67.2 us, 14.4 sy,  0.0 ni,  2.0 id,  0.0 wa,  0.0 hi, 16.4 si,  0.0 s
%Cpu1  : 66.7 us, 18.4 sy,  0.0 ni, 11.9 id,  0.0 wa,  0.0 hi,  3.1 si,  0.0 s
KiB Mem :  8008964 total,   267664 free,  2270248 used,  5471052 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  5430304 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13465 root      20   0 4607344 491400  12044 S  94.0  6.1  18:21.24 java -Dcom
 7931 root      20   0 4636424 713848  11920 S  64.1  8.9 120:26.35 java -Dcom
 1473 root      20   0  675856  84604  20440 S  14.6  1.1 562:05.09 /usr/bin/d
```

```
 1894 root      20   0 1729636  83780  24696 S   3.7  1.0 649:29.87 /usr/bin/k
13446 root      20   0  109096   3724   2768 S   2.3  0.0   0:23.05 containerd
 7913 root      20   0  109096   5684   2716 S   2.0  0.1   4:42.36 containerd
21598 root      10 -10  229756  58396  13980 S   2.0  0.7  32:35.85 /usr/local
```

从上面的数据可以看到，有两个 Java 进程消耗 CPU 比较多。这个时候我们要再往下分析就需要考虑是从哪里开始了。上面的数据中，us CPU 在 66% 左右，sy 在 20% 以下，这两个部分是挺正常的数据了。但 si CPU 达到了 16%，这就有点高了。我们可以先从这里往下找，查看软中断计数器（注意这个值是累加值）：

复制代码

```
[root@s6 ~]# cat /proc/softirqs
                    CPU0       CPU1
          HI:          1          0
       TIMER:  234909974  214047397
      NET_TX:         25         40
      NET_RX:  126031087   17310460
       BLOCK:     904406          0
BLOCK_IOPOLL:          0          0
     TASKLET:     175969        485
       SCHED:  113910312  103576801
     HRTIMER:          0          0
         RCU:  169468493  162852369
```

可以看到，CPU0 确实比 CPU1 中断次数要多很多。

我们接着查看一下网卡队列。这里需要查看两个网卡，一个是系统网卡，一个是 k8s 网络插件 Flannel 的网卡。

复制代码

```
[root@s6 ~]# ll /sys/class/net/eth0/queues/
total 0
drwxr-xr-x 2 root root 0 Nov 17 23:33 rx-0
drwxr-xr-x 2 root root 0 Nov 17 23:33 rx-1
drwxr-xr-x 3 root root 0 Nov 17 23:33 tx-0
drwxr-xr-x 3 root root 0 Nov 17 23:33 tx-1
[root@s6 ~]# ll /sys/class/net/flannel.1/queues/
total 0
drwxr-xr-x 2 root root 0 Nov 17 15:47 rx-0
drwxr-xr-x 3 root root 0 Nov 17 15:47 tx-0
[root@s6 ~]#
```

从检查的结果来看，系统的网卡队列是和 CPU 数一致的，而 Flannel 的插件只创建了一个网卡接收队列和一个网卡发送队列。

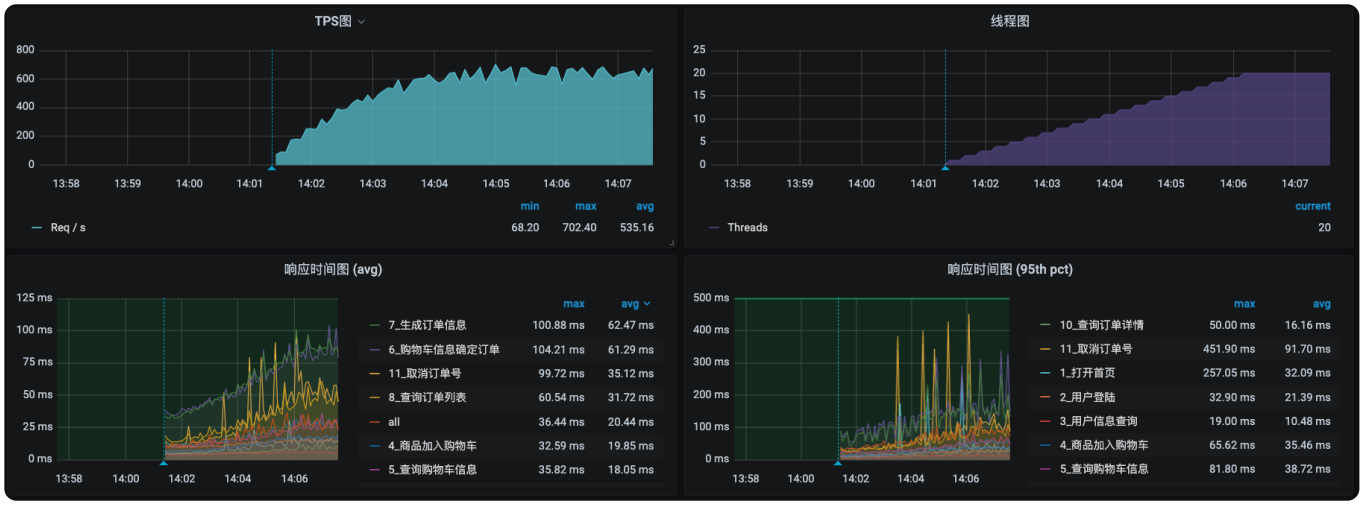
我们再查一下这上面总共有多少个 Pod。

复制代码

```
[root@s5 runMall]# kubectl get pods -n default -o wide|grep s6
kafka-1                                  1/1     Running     48         13d
node-exporter-rqgrx                      1/1     Running     0          3d21h
svc-mall-cart-dd5db86d7-dcwdd            1/1     Running     0          4d16h
svc-mall-member-85668c9bbf-f97kg         1/1     Running     0          16h
[root@s5 runMall]#
```

从查询的结果来看，这里有 Cart 和 Member 两个服务，还有 Kafka 和 node_exporter，这是每个机器上都会有的监控容器。

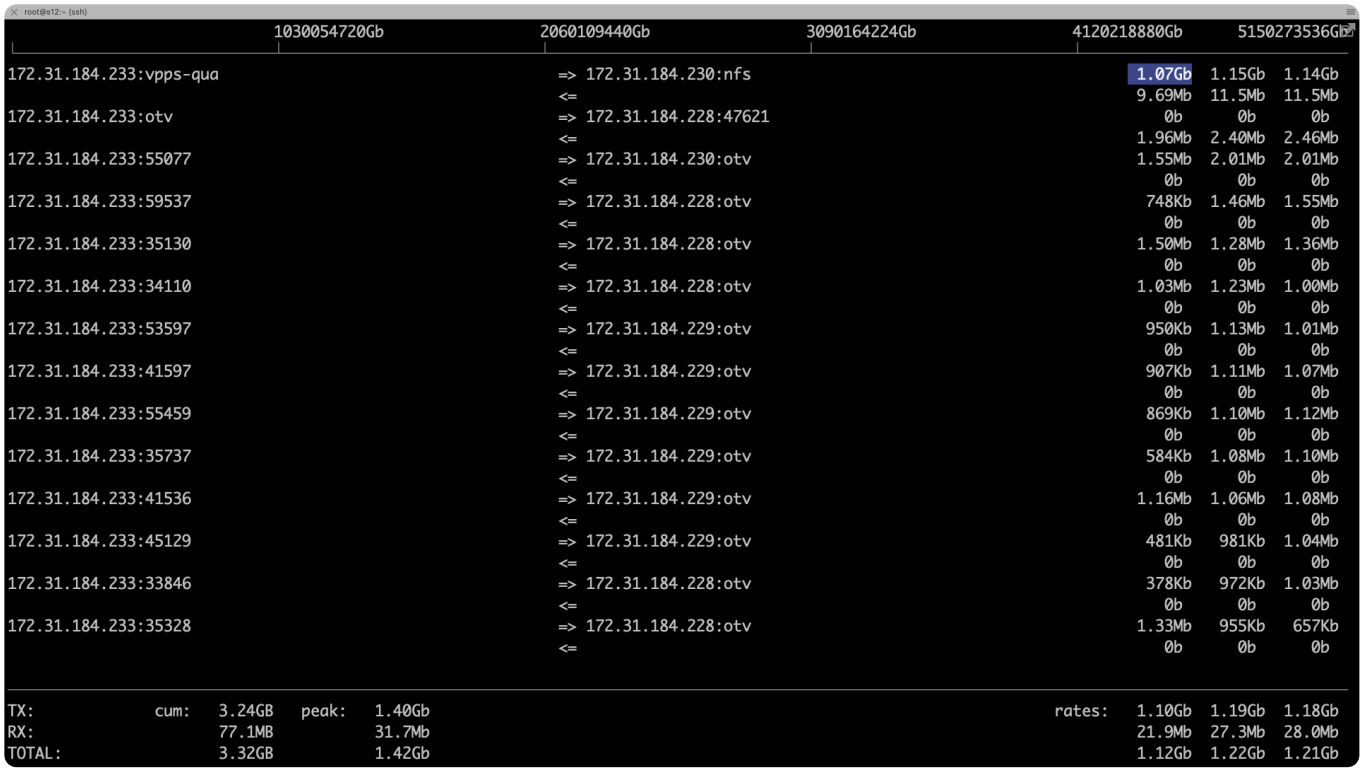
因为 s6 只有 2C，数据即使能接收下来处理能力也有限，所以我们先把服务从 s6 移到其他主机上去，然后再执行一下场景，看看效果：



这次 TPS 增加到了 700 左右，已经不再增加了。主机资源呢，s12 的 CPU 使用率达到了 75% 左右。 同时，网络带宽也用起来了。

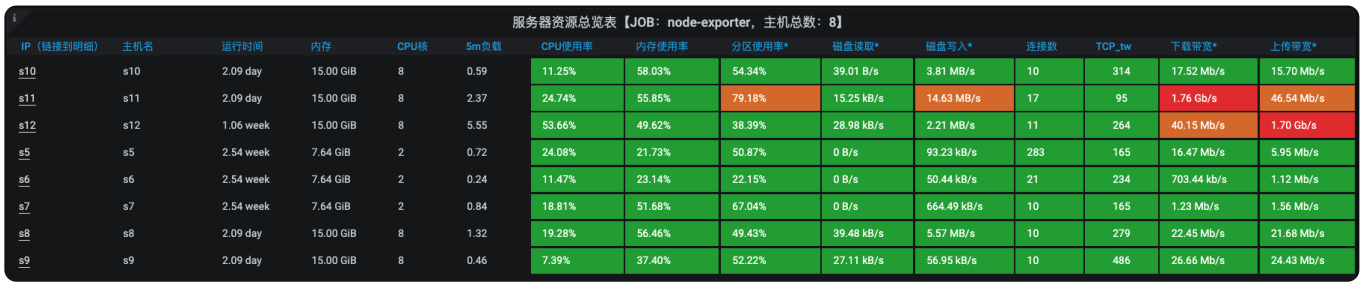
IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	6.31 hour	15.00 GiB	8	0.44	12.84%	56.38%	55.34%	554.87 kB/s	4.09 MB/s	10	286	17.09 Mb/s	15.32 Mb/s
s11	s11	6.31 hour	15.00 GiB	8	0.85	23.72%	55.41%	77.39%	602.19 kB/s	10.91 MB/s	17	73	1.31 Gb/s	39.92 Mb/s
s12	s12	5.57 day	15.00 GiB	8	5.46	72.38%	50.08%	34.06%	741.18 B/s	102.44 kB/s	11	282	114.15 Mb/s	1.26 Gb/s
s5	s5	2.28 week	7.64 GiB	2	0.61	19.24%	31.46%	50.85%	0 B/s	82.08 kB/s	293	151	14.17 Mb/s	4.68 Mb/s
s6	s6	2.28 week	7.64 GiB	2	0.38	10.38%	22.55%	21.40%	0 B/s	7.41 kB/s	21	267	511.88 kb/s	862.36 kb/s
s7	s7	2.28 week	7.64 GiB	2	0.39	19.33%	73.82%	73.46%	78.02 B/s	761.66 kB/s	10	237	1.23 Mb/s	1.54 Mb/s
s8	s8	6.31 hour	15.00 GiB	8	0.85	17.58%	46.39%	43.36%	532.21 kB/s	5.81 MB/s	10	235	20.92 Mb/s	20.55 Mb/s
s9	s9	6.31 hour	15.00 GiB	8	0.66	14.85%	34.75%	56.28%	39.01 B/s	5.34 MB/s	10	446	25.03 Mb/s	22.91 Mb/s

不过这个网络带宽还是有点高的，所以我们来查一下是什么消耗了这么大的网络带宽。登录到 s11 上，执行下 iftop 命令。

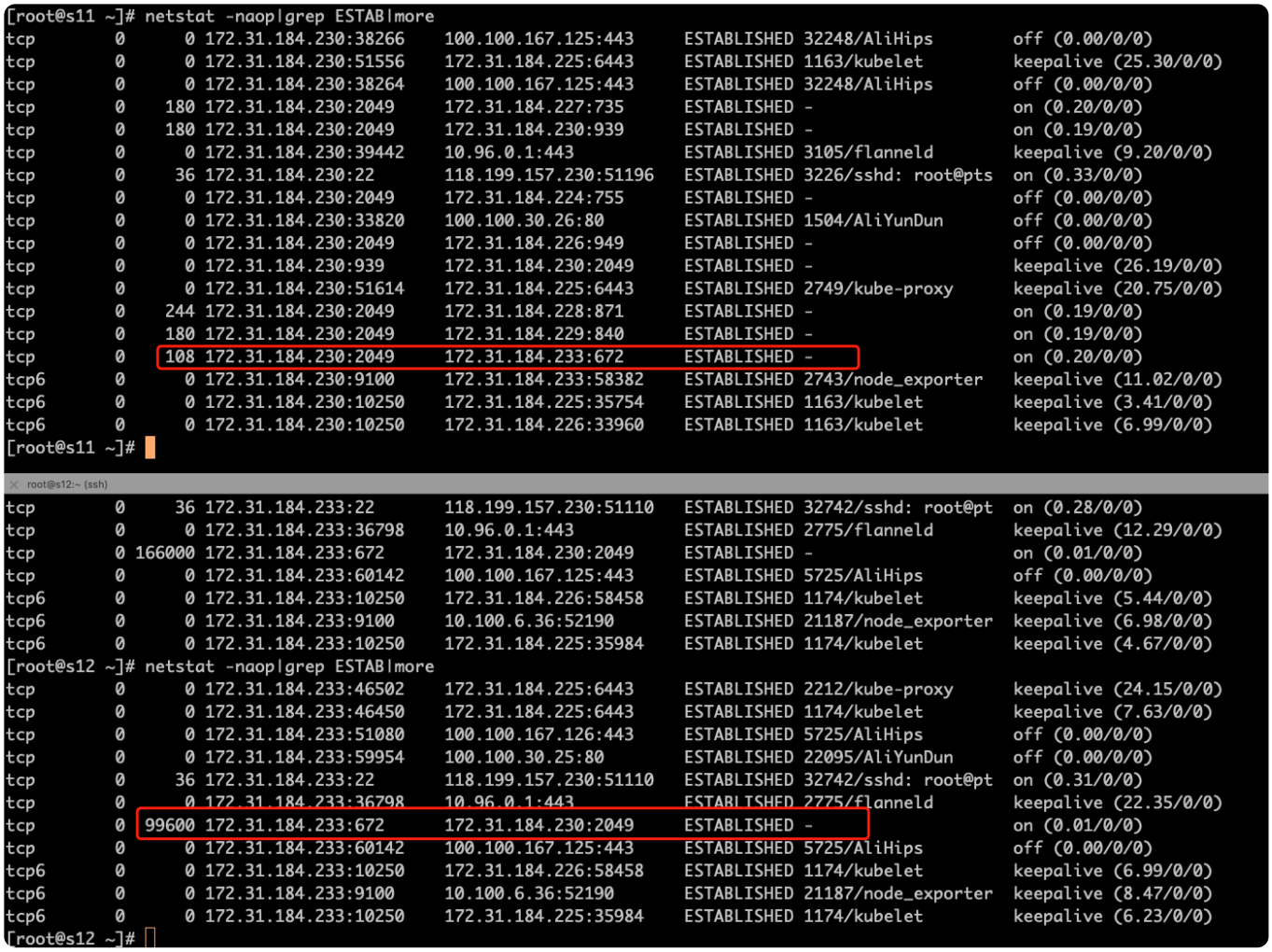


发现这是 NFS 做同步时使用的带宽。172.31.184.233 这台主机是 s12，而 s12 上使用最多的是我们的 shadow 数据库。

从全局监控的数据来看，s12 和 s11 之间的带宽占得实在是有点高。在同步比较大的时候，居然能达到 1.7G 以上。就像下面这张图：



我们到 s12 和 s11 上分别执行命令 netstat，查看一下网络的队列是否存在。



这一步我们要把两边的机器对应着来看。显然，这里是长期有网络队列的，我多次执行了命令来查看这个数据，发现双方还都有 send_Q 队列值。这就明显是网络带宽不足的问题了。

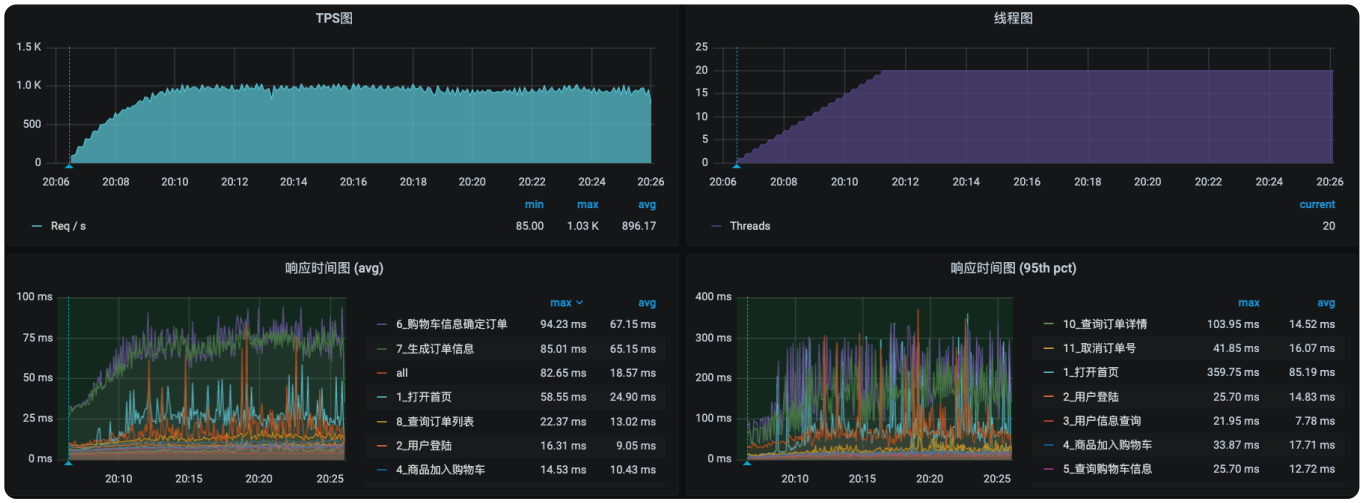
因为 s12 上是数据库，现在的问题就是，数据库作持久化时，由于数据量较大，所以把持久化数据通过 NFS 写到 s11 上会占用很大的带宽。要想让带宽降下来，只有一个办法就是把 MySQL 表数据大小降下来。

在这里，我先把压力产生的数据库的数据给清理一下，像记录表、订单表、购物车表等等，让 NFS 的流量先降下来。然后再来看看 TPS 能不能上升一些。

复制代码

```
truncate table mall.oms_order_item ;
truncate table mall.ums_member_login_log ;
truncate table mall.oms_cart_item ;
analyze table mall.oms_order_item ;
analyze table mall.ums_member_login_log ;
analyze table mall.oms_cart_item ;
```

一顿迅猛的操作之后，我们再来看看压力数据。



哟，这一下可是快到 1000TPS 了！再来看一下全局监控的数据。

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	2.53 day	15.00 GiB	8	1.39	13.98%	56.29%	63.52%	19.15 kB/s	6.91 MB/s	10	287	19.11 Mb/s	17.21 Mb/s
s11	s11	2.53 day	15.00 GiB	8	2.07	19.68%	55.43%	72.45%	49.89 kB/s	5.24 MB/s	18	79	229.04 Mb/s	32.68 Mb/s
s12	s12	1.12 week	15.00 GiB	8	5.10	40.10%	47.49%	49.53%	780.19 B/s	2.32 MB/s	10	300	129.12 Mb/s	118.36 Mb/s
s5	s5	2.60 week	7.64 GiB	2	0.71	21.99%	32.88%	50.90%	0 B/s	221.57 kB/s	310	172	14.20 Mb/s	5.12 Mb/s
s6	s6	2.60 week	7.64 GiB	2	0.30	12.10%	23.00%	22.39%	0 B/s	45.17 kB/s	22	252	2.70 Mb/s	3.72 Mb/s
s7	s7	2.60 week	7.64 GiB	2	0.70	19.53%	51.98%	70.63%	429.10 B/s	952.38 kB/s	10	141	1.40 Mb/s	1.72 Mb/s
s8	s8	2.53 day	15.00 GiB	8	2.50	18.61%	54.70%	23.33%	380.07 kB/s	6.79 MB/s	10	109	20.73 Mb/s	21.31 Mb/s
s9	s9	2.53 day	15.00 GiB	8	3.64	28.46%	38.41%	54.98%	39.01 B/s	71.35 kB/s	23	413	121.05 Mb/s	298.69 Mb/s

网络带宽也已经降下来很多了。 效果看起来还不错！

不过这个优化动作虽然在我们这个场景中有效，但是不是一个一劳永逸的解决办法。正常的处理方式应该是换个性能好的 IO 存储设备，我们这个云服务器的本地磁盘做的 NFS，确实在读写能力上是比较差的。当然啦，我们这个示例项目还是可以继续通过清理数据的方式先往下分析。

第二阶段

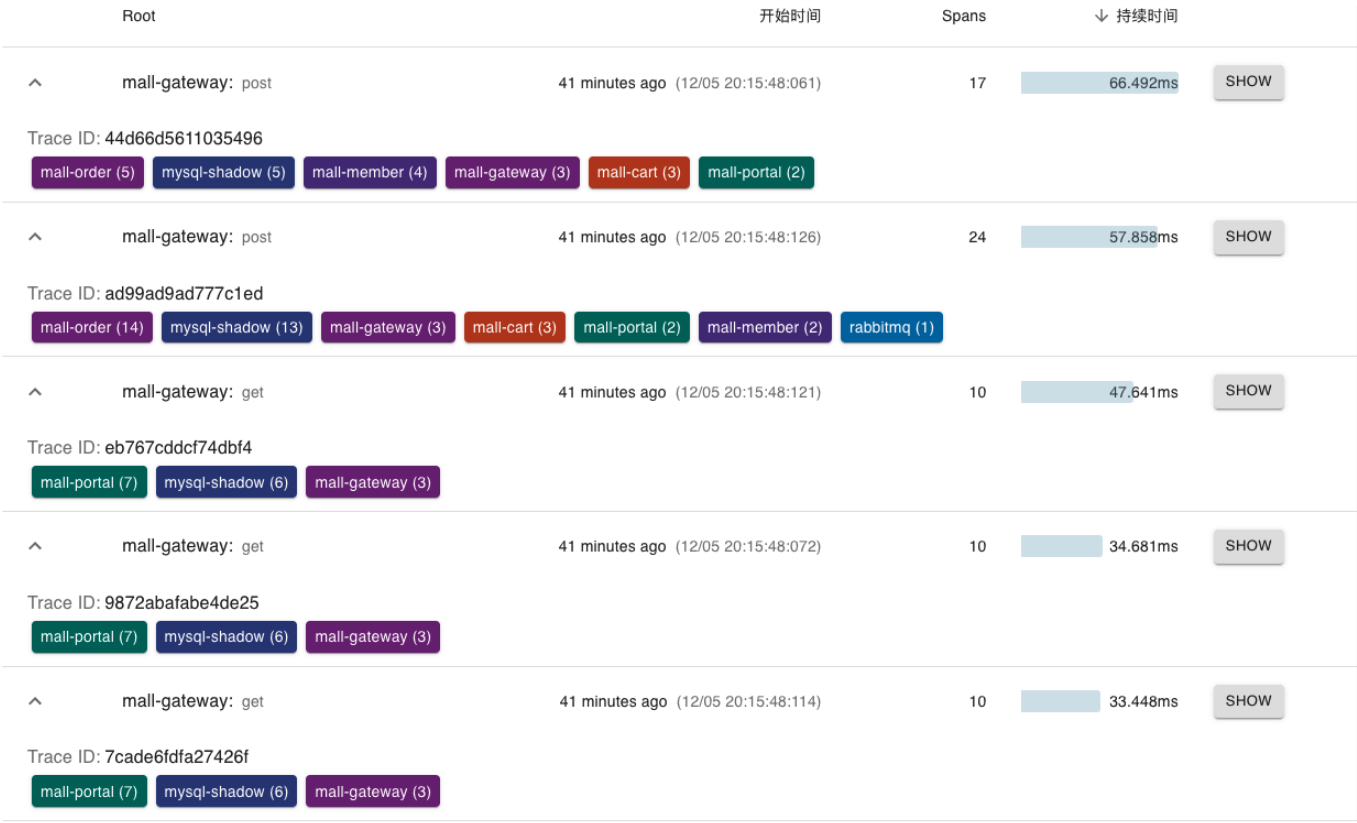
从上面的监控截图中可以看到，我们的目标 TPS 还没有达到，响应时间倒是挺低。各服务器的 CPU 资源使用率也并没有特别高，资源都没有用完，还是心有不甘的。

接下来应该怎么办呢？我们还是要来分析有没有优化的点。

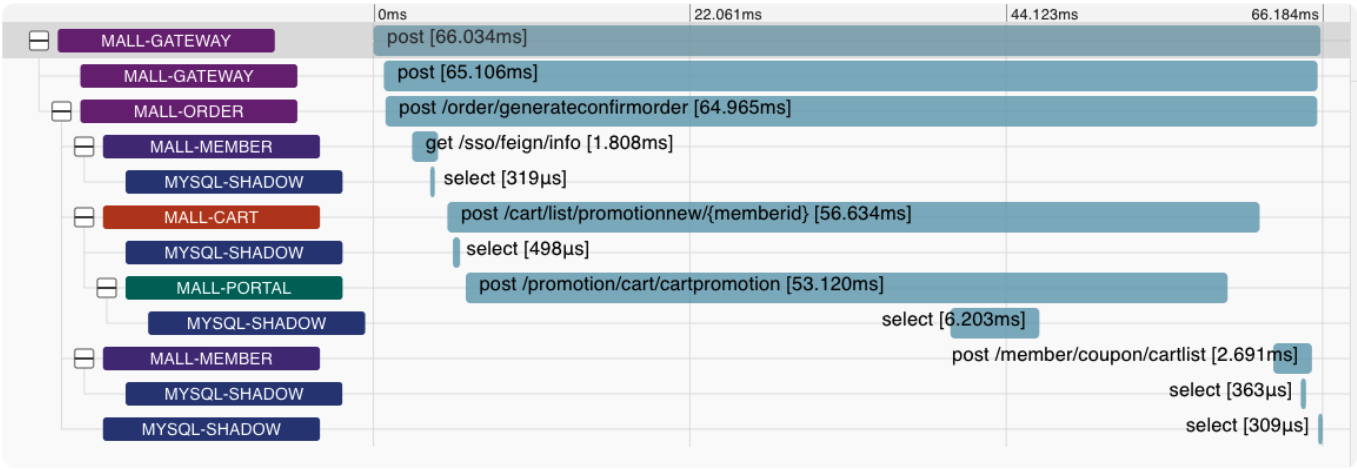
拆分响应时间

先来看看响应时间。我们得想想时间消耗到哪里去了，再来判断下有没有优化时间的可能性，从资源没有用完的角度来看，应该是有阻塞点。

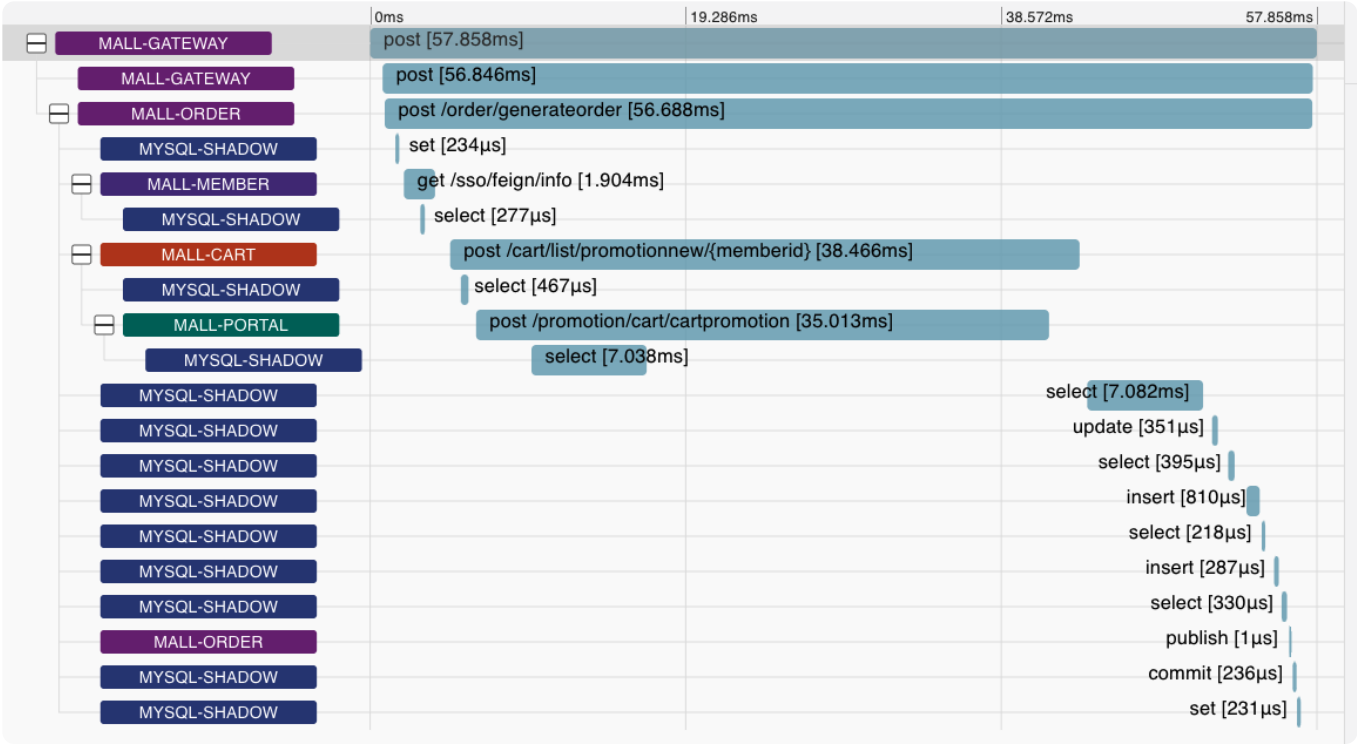
打开 Zipkin，搜索一下结果：



这个时间倒是和 JMeter 上的时间是接近的。我们点第一个接口 show 看看：



再点第二个接口 show，看到的是下面的结果：



从这两个接口的调用树来看，都是在 cartpromotion 这里比较慢，子调用中的 SQL 语句看起来也并不慢。

定向监控分析

接着我们就要看看是哪个具体的方法耗时比较长了。这里我用 Arthas 跟踪一下 cartpromotion 接口。

我们可以先从代码中找到 cartpromotion 的接口代码：

复制代码

```
    @PostMapping("/cart/CartPromotion")
    @ResponseBody
    List<CartPromotionItem> calcCartPromotion(@RequestBody List<OmsCartItem> c
        List<CartPromotionItem> cartPromotionItems = omsPromotionService.calcC
        return cartPromotionItems;
    }
```

这里是 calcCartPromotion，是计算优惠券的。又因为这个接口是从 Portal 里面调的，所以我们进入 Portal 服务中，下载 Arthas 并启动跟踪。

复制代码

```
[root@svc-mall-portal-75cf5c7989-ccsr4 /]# curl -O https://arthas.aliyun.com/a
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Curren
                                 Dload  Upload   Total   Spent    Left  Speed
100  138k  100  138k    0     0   363k      0 --:--:-- --:--:-- --:--:--  364k
[root@svc-mall-portal-75cf5c7989-ccsr4 /]# java -jar arthas-boot.jar

[arthas@1]$ trace com.dunshan.mall.portal.controller.CartPromotionController c
Press Q or Ctrl+C to abort.
Affect(class count: 1 , method count: 1) cost in 97 ms, listenerId: 6
Condition express: 1==1 , result: true
`---ts=2021-12-05 21:50:29;thread_name=http-nio-8085-exec-85;id=74b5;is_daemon
    `---[37.443819ms] com.dunshan.mall.portal.controller.CartPromotionControll
        `---[37.38849ms] com.dunshan.mall.portal.service.OmsPromotionService:c

Condition express: 1==1 , result: true
`---ts=2021-12-05 21:50:29;thread_name=http-nio-8085-exec-73;id=7438;is_daemon
    `---[34.832809ms] com.dunshan.mall.portal.controller.CartPromotionControll
        `---[34.811947ms] com.dunshan.mall.portal.service.OmsPromotionService:
Condition express: 1==1 , result: true

`---ts=2021-12-05 21:50:29;thread_name=http-nio-8085-exec-86;id=74b9;is_daemon

Condition express: 1==1 , result: true
`---ts=2021-12-05 21:50:29;thread_name=http-nio-8085-exec-81;id=7480;is_daemon

    `---[41.252226ms] com.dunshan.mall.portal.controller.CartPromotionControll
        `---[40.220118ms] com.dunshan.mall.portal.service.OmsPromotionService:

`---ts=2021-12-05 21:50:29;thread_name=http-nio-8085-exec-71;id=7430;is_daemon
Command execution times exceed limit: 5, so command will exit. You can set it
    `---[42.934061ms] com.dunshan.mall.portal.controller.CartPromotionControll
        `---[42.849213ms] com.dunshan.mall.portal.service.OmsPromotionService:
```

接着跟踪。

复制代码

```
[arthas@1]$ trace com.dunshan.mall.portal.service.OmsPromotionService calcCart
Condition express: 1==1 , result: true
`---ts=2021-12-05 21:51:36;thread_name=http-nio-8085-exec-91;id=74db;is_daemon
    +---[0.007964ms] com.dunshan.mall.portal.service.impl.OmsPromotionServiceI
    +---[19.243656ms] com.dunshan.mall.portal.service.impl.OmsPromotionService
    +---[0.002098ms] java.util.ArrayList:<init>() #44
    +---[0.001859ms] java.util.Map:entrySet() #45
    +---[0.002208ms] java.util.Set:iterator() #45
    +---[min=0.001209ms,max=0.001753ms,total=0.002962ms,count=2] java.util.Ite
    +---[0.001215ms] java.util.Iterator:next() #45
    +---[0.001574ms] java.util.Map$Entry:getKey() #46
    +---[0.003588ms] com.dunshan.mall.portal.service.impl.OmsPromotionServiceI
    +---[0.004646ms] java.util.Map$Entry:getValue() #50
    +---[0.001664ms] com.dunshan.mall.portal.domain.PromotionProduct:getPromot
```

```
    `---[0.038219ms] com.dunshan.mall.portal.service.impl.OmsPromotionServiceI

Condition express: 1==1 , result: true
`---ts=2021-12-05 21:51:36;thread_name=http-nio-8085-exec-85;id=74b5;is_daemon
    `---[40.440293ms] com.dunshan.mall.portal.service.impl.OmsPromotionService
        +---[0.010603ms] com.dunshan.mall.portal.service.impl.OmsPromotionServ
        +---[18.010145ms] com.dunshan.mall.portal.service.impl.OmsPromotionSer
        +---[0.002138ms] java.util.ArrayList:<init>() #44
        +---[0.001818ms] java.util.Map:entrySet() #45
        +---[0.001696ms] java.util.Set:iterator() #45
        +---[min=0.00118ms,max=0.001659ms,total=0.002839ms,count=2] java.util.
        +---[0.0012ms] java.util.Iterator:next() #45
        +---[0.00145ms] java.util.Map$Entry:getKey() #46
        +---[0.004004ms] com.dunshan.mall.portal.service.impl.OmsPromotionServ
        +---[0.001356ms] java.util.Map$Entry:getValue() #50
        +---[0.001632ms] com.dunshan.mall.portal.domain.PromotionProduct:getPr
        `---[0.043778ms] com.dunshan.mall.portal.service.impl.OmsPromotionServ
```

接着跟踪。

复制代码

```
Condition express: 1==1 , result: true
`---ts=2021-12-05 21:58:38;thread_name=http-nio-8085-exec-94;id=7551;is_daemon
    `---[18.520576ms] com.dunshan.mall.portal.service.impl.OmsPromotionService
        +---[0.001613ms] java.util.ArrayList:<init>() #145
        +---[0.001367ms] java.util.List:iterator() #146
        +---[min=0.001099ms,max=0.001269ms,total=0.002368ms,count=2] java.util
        +---[0.001267ms] java.util.Iterator:next() #146
        +---[0.001399ms] com.dunshan.mall.model.OmsCartItem:getProductId() #14
        +---[0.00125ms] java.util.List:add() #147
        `---[17.622753ms] com.dunshan.mall.portal.dao.PortalProductDao:getProm

Condition express: 1==1 , result: true
`---ts=2021-12-05 21:58:38;thread_name=http-nio-8085-exec-109;id=7576;is_daemo
    `---[58.172854ms] com.dunshan.mall.portal.service.impl.OmsPromotionService
        +---[0.002119ms] java.util.ArrayList:<init>() #145
        +---[0.001274ms] java.util.List:iterator() #146
        +---[min=0.001152ms,max=0.001362ms,total=0.002514ms,count=2] java.util
        +---[0.001101ms] java.util.Iterator:next() #146
        +---[0.001395ms] com.dunshan.mall.model.OmsCartItem:getProductId() #14
        +---[0.001216ms] java.util.List:add() #147
        `---[58.125958ms] com.dunshan.mall.portal.dao.PortalProductDao:getProm
```

可以看到，跟踪到 DAO 层的时候，这里就直接对应着一个 SQL 了。

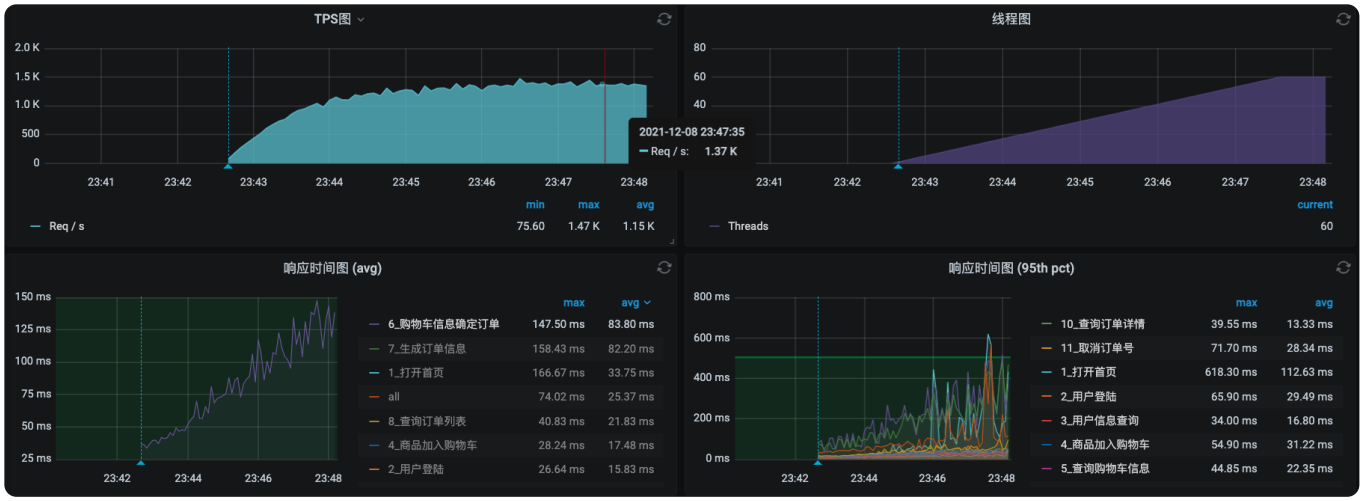
复制代码

```
SELECT
    p.id,
    p.`name`,
    p.promotion_type,
    p.gift_growth,
    p.gift_point,
    sku.id sku_id,
    sku.price sku_price,
    sku.sku_code sku_sku_code,
    sku.promotion_price sku_promotion_price,
    sku.stock sku_stock,
    sku.lock_stock sku_lock_stock,
    ladder.id ladder_id,
    ladder.count ladder_count,
    ladder.discount ladder_discount,
    full_re.id full_id,
    full_re.full_price full_full_price,
    full_re.reduce_price full_reduce_price
FROM
    mall.pms_product p
        LEFT JOIN
    mall.pms_sku_stock sku ON p.id = sku.product_id
        LEFT JOIN
    mall.pms_product_ladder ladder ON p.id = ladder.product_id
        LEFT JOIN
    mall.pms_product_full_reduction full_re ON p.id = full_re.product_id
WHERE
    p.id IN (7);
```

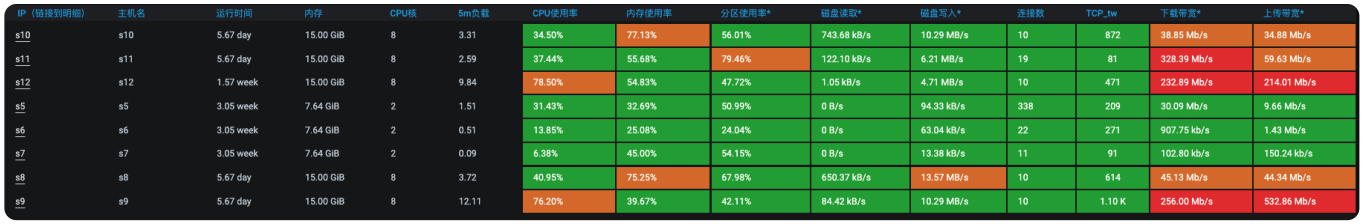
可是要说 SQL 执行效果差，也没见数据库所在的节点资源使用率高。这时就不能不来查查 JDBC 连接池了：

```
characterEncoding=utf-8&serverTimezone=Asia/Shanghai&allowMultiQueries=true
    username: reader
    password:
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      initial-size: 10 #连接池初始化大小
      min-idle: 10 #最小空闲连接数
      max-active: 20 #最大连接数
      max-wait: 60000  # 配置获取连接等待超时的时间
      time-between-eviction-runs-millis: 60000  # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
      min-evictable-idle-time-millis: 300000  # 指定一个空闲连接最少空闲多久后可被清除，单位是毫秒
      validationQuery: select 'x'
      test-while-idle: true  # 当连接空闲时，是否执行连接测试
      test-on-borrow: false  # 当从连接池借用连接时，是否测试该连接
      test-on-return: false  # 在连接归还到连接池时是否测试该连接
      web-stat-filter:
        exclusions: "*.js,*.gif,*.jpg,*.png,*.css,*.ico,/druid/*" #不统计这些请求数据
```

从这张 application.yml 截图里可以看到，最大 JDBC 也只有 20，有点低了。我们调到 40，把压力线程也提高到 60，再压起来，看一下压力场景的数据：



这时候 TPS 已经能达到 1300 了！可以达到我们对这个系统的 TPS 的预期目标了。再来看一下主机的资源：



IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	5.67 day	15.00 GiB	8	3.31	34.50%	77.13%	56.01%	743.68 kB/s	10.29 MB/s	10	872	38.85 Mb/s	34.88 Mb/s
s11	s11	5.67 day	15.00 GiB	8	2.59	37.44%	55.68%	79.46%	122.10 kB/s	6.21 MB/s	19	81	328.39 Mb/s	59.63 Mb/s
s12	s12	1.57 week	15.00 GiB	8	9.84	78.50%	54.83%	47.72%	1.05 kB/s	4.71 MB/s	10	471	232.89 Mb/s	214.01 Mb/s
s5	s5	3.05 week	7.64 GiB	2	1.51	31.43%	32.69%	50.99%	0 B/s	94.33 kB/s	338	209	30.09 Mb/s	9.66 Mb/s
s6	s6	3.05 week	7.64 GiB	2	0.51	13.85%	25.08%	24.04%	0 B/s	63.04 kB/s	22	271	907.75 kb/s	1.43 Mb/s
s7	s7	3.05 week	7.64 GiB	2	0.09	6.38%	45.00%	54.15%	0 B/s	13.38 kB/s	11	91	102.80 kb/s	150.24 kb/s
s8	s8	5.67 day	15.00 GiB	8	3.72	40.95%	75.25%	67.98%	650.37 kB/s	13.57 MB/s	10	614	45.13 Mb/s	44.34 Mb/s
s9	s9	5.67 day	15.00 GiB	8	12.11	76.20%	39.67%	42.11%	84.42 kB/s	10.29 MB/s	10	1.10 K	256.00 Mb/s	532.86 Mb/s

挺好！资源也用起来了。

不过这里还要说明一下，calcCartPromotion 方法的实现是比较复杂的。代码你可以看看我给你截的下面这张图：

```
public List<CartPromotionItem> calcCartPromotion(List<OmsCartItem> cartItemList) {
    //1.先根据productId对CartItem进行分组，以spu为单位进行计算优惠
    Map<Long, List<OmsCartItem>> productCartMap = groupCartItemBySpu(cartItemList);
    //2.查询所有商品的优惠相关信息
    List<PromotionProduct> promotionProductList = getPromotionProductList(cartItemList);
    //3.根据商品促销类型计算商品促销优惠价格
    List<CartPromotionItem> cartPromotionItemList = new ArrayList<>();
    for (Map.Entry<Long, List<OmsCartItem>> entry : productCartMap.entrySet()) {
        Long productId = entry.getKey();
        PromotionProduct promotionProduct = getPromotionProductById(productId, promotionProductList);
        if (promotionProduct == null)
            continue;
        List<OmsCartItem> itemList = entry.getValue();
        Integer promotionType = promotionProduct.getPromotionType();
        if (promotionType == 1) {...} else if (promotionType == 3) {
            //打折优惠
            int count = getCartItemCount(itemList);
            PmsProductLadder ladder = getProductLadder(count, promotionProduct.getProductLadderList());
            if(ladder!=null){...}else{
                handleNoReduce(cartPromotionItemList,itemList,promotionProduct);
            }
        } else if (promotionType == 4) {
            //满减
            BigDecimal totalAmount= getCartItemAmount(itemList,promotionProductList);
            PmsProductFullReduction fullReduction = getProductFullReduction(totalAmount,promotionProduct.getProductFullReductionList());
            if(fullReduction!=null){...}else{
                handleNoReduce(cartPromotionItemList,itemList,promotionProduct);
            }
        } else {
            //无优惠
            handleNoReduce(cartPromotionItemList, itemList,promotionProduct);
        }
    }
    return cartPromotionItemList;
}
```

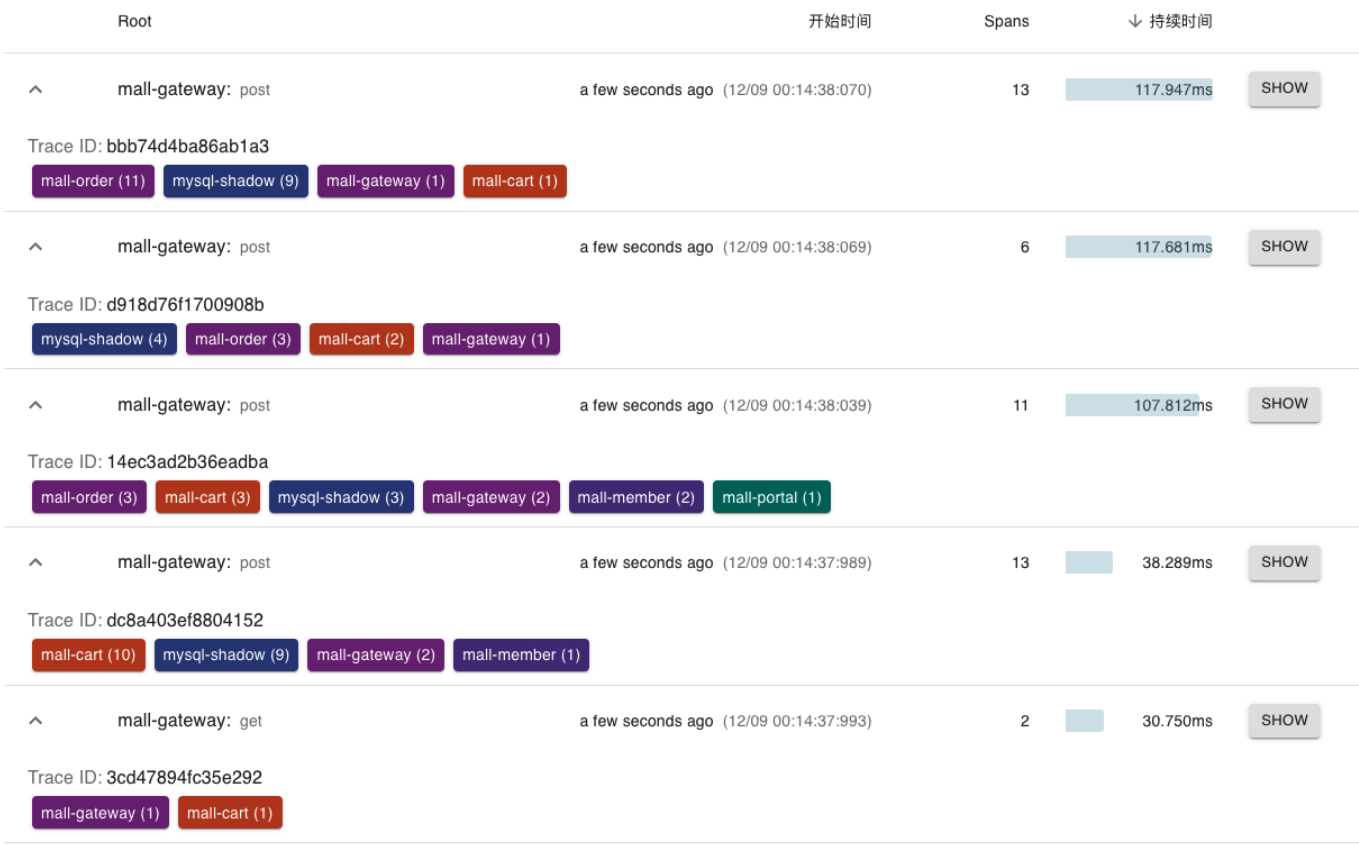
这还是折叠起来的。如果把代码都展开，一屏都显示不完。如果想再优化的话，就得考虑一下怎么优化这个方法了。

因为我们现在要做的是全链路压测，现在我们的主要目标是跑出正常流量和压测流量来，所以这个优化方向就先放一放，先跑出两种流量的场景再说。

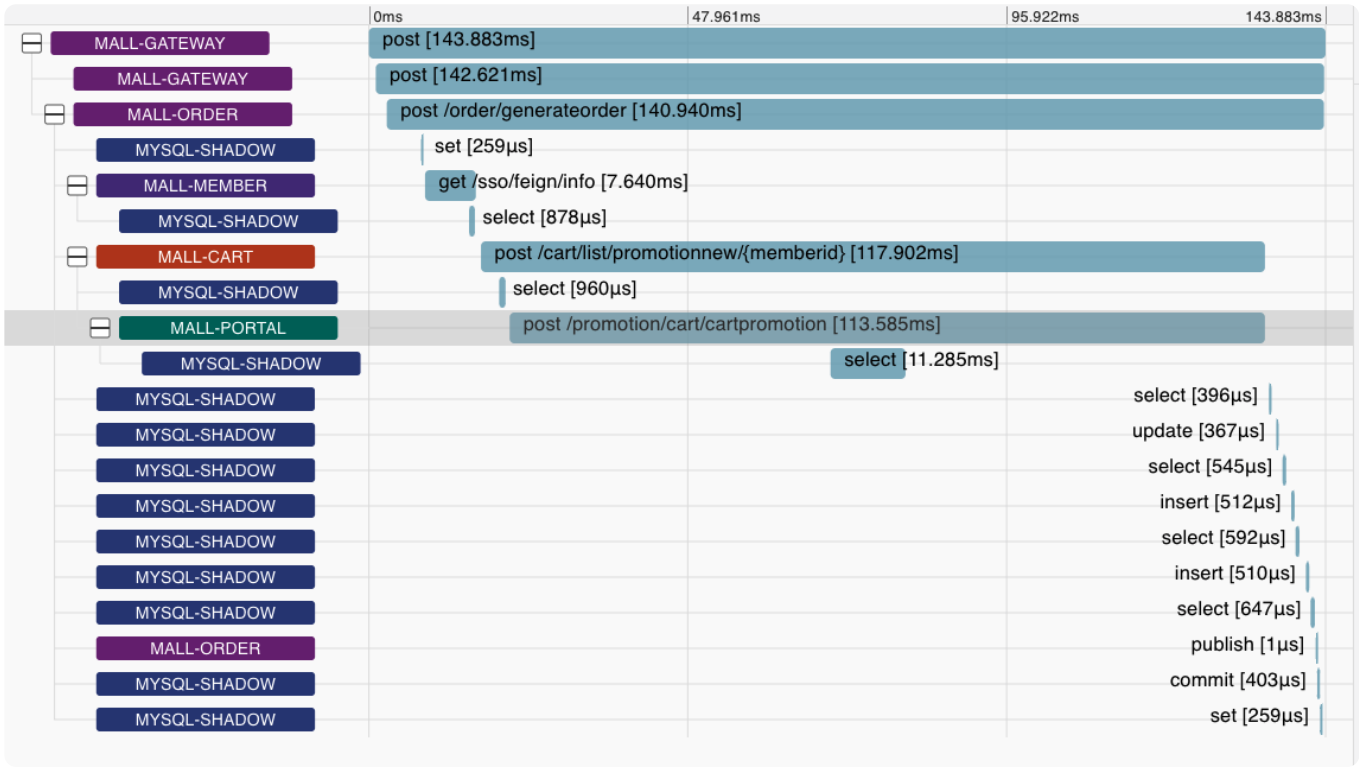
第三阶段

我们再接着来分析还有没有优化的空间。既然刚才我们已经看到了，有些接口的响应时间会随着压力增加而增加，那么我们就直接来判断一下响应时间消耗在哪里去了。

打开 Zipkin，看一下排名前十的请求，这里我截取几个图给你看一下。



点开一个来看一下。

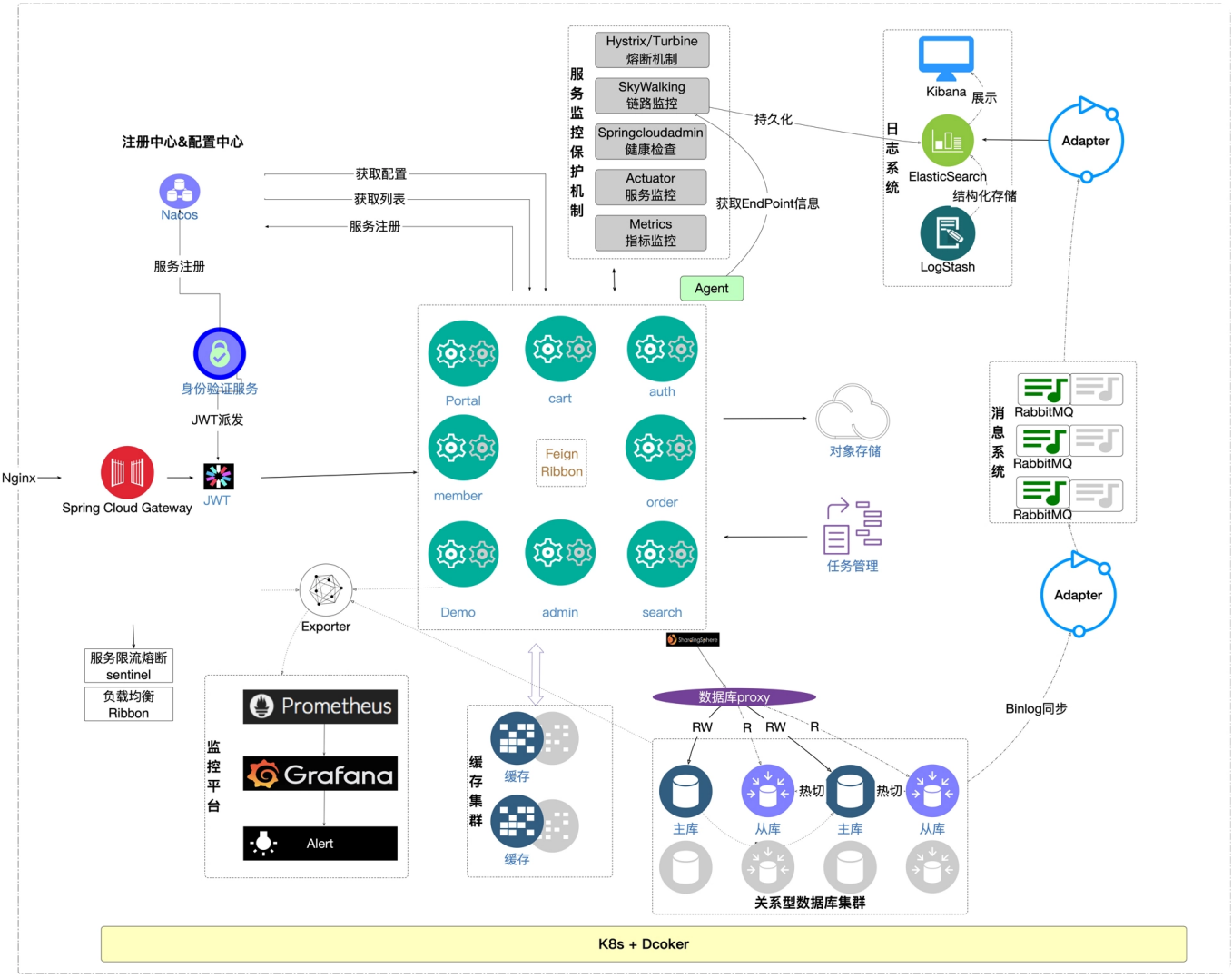


可以看到，这里 cartpromotion 耗时是最长的，这个逻辑我们前面已经看过了，我就不重复上面的动作了。

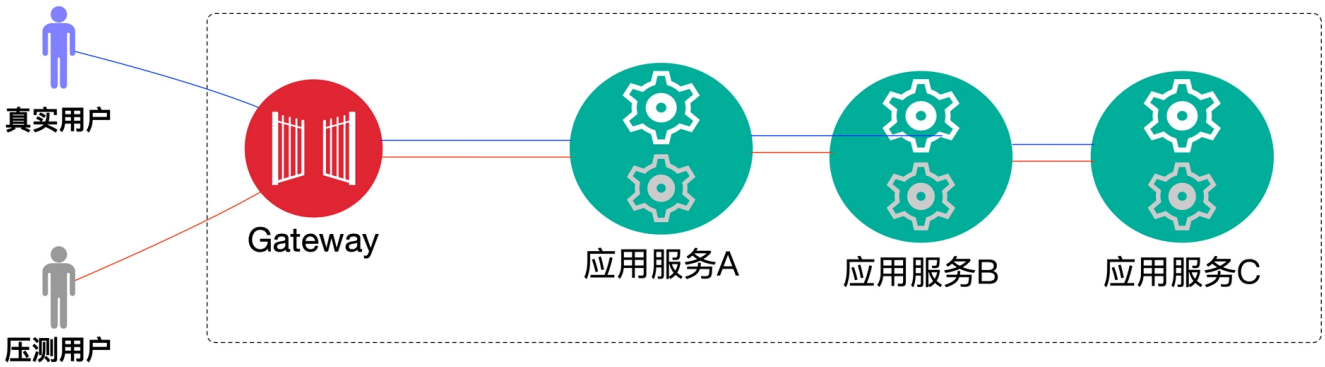
同时启动正常流量和压测流量

现在我们换个逻辑来跑。我们一边用 JMeter 来跑正常的流量，一边用 GoReplay 来跑压测的流量。

还记得我们前面的全链路改造架构图吧。



我把它简化一下就是下面的样子：



现在我们就来按全链路压测的逻辑来实现真实流量和压测流量。我们把 JMeter 运行起来，TPS 在 100 左右，GoReplay 放大 150 倍。因为 GoReplay 本身没办法实现连续递增加压（可以自己写脚本实现），我们这里就直接加压力了。

先看看看能达到多高的峰值，我们前面知道，系统的整体 TPS 大概在 1300 左右。

从上面的截图可以看出来，左边的 GoReplay 运行了两个实例，加在一起大概能达到 1250 左右，右边 JMeter 大概能达到 120TPS 左右。两边加在一起在 1370 左右。运行非常稳定。

再来看看主机资源使用情况。

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
s10	s10	1.19 week	15.00 GiB	8	1.73	19.25%	67.00%	69.70%	48.33 kB/s	4.86 MB/s	11	398	28.09 Mb/s	25.11 Mb/s
s11	s11	1.19 week	15.00 GiB	8	14.29	78.24%	70.58%	91.99%	647.75 kB/s	21.36 MB/s	21	1.09 K	269.75 Mb/s	89.53 Mb/s
s12	s12	1.95 week	15.00 GiB	8	2.08	14.83%	52.11%	44.04%	41.23 kB/s	3.90 MB/s	10	342	31.51 Mb/s	36.19 Mb/s
s5	s5	3.43 week	7.64 GiB	2	0.87	25.13%	32.19%	50.82%	0 B/s	119.06 kB/s	295	178	18.50 Mb/s	6.30 Mb/s
s6	s6	3.43 week	7.64 GiB	2	1.89	28.82%	26.16%	25.45%	0 B/s	65.03 kB/s	34	325	16.85 Mb/s	6.60 Mb/s
s7	s7	3.43 week	7.64 GiB	2	0.69	20.61%	50.09%	55.17%	0 B/s	759.16 kB/s	11	139	5.05 Mb/s	4.64 Mb/s
s8	s8	1.19 week	15.00 GiB	8	4.19	47.62%	70.63%	70.48%	177.61 kB/s	10.13 MB/s	10	1.30 K	63.57 Mb/s	57.59 Mb/s
s9	s9	1.19 week	15.00 GiB	8	3.70	38.67%	43.91%	45.65%	155.96 kB/s	6.34 MB/s	10	391	82.21 Mb/s	238.42 Mb/s

现在 s11 这个主机，CPU 使用率接近 80%，其他的还好，说明硬件资源还有空间。

再来看一下两个数据库的流量。

复制代码

```
[root@s5 ~]# kubectl get pods -n default -o wide | grep mysql
db-mysql-produce-7b8fc76c76-45nc7          2/2     Running            5
db-mysql-shadow-74ff9bdfb4-kkxbq           2/2     Running            0
```

```
4 [root@s5 ~]#
```

这是两个数据库的信息，我们把两边的监控都打开对比看一下。

可以看到，影子库的连接（MySQL Connections 图）和执行的 SQL 数量（MySQL Questions 图）都比正式库要高出很多。

作为全链路压测的场景，这样的资源使用率是比较合理的。因为我们不能给生产环境带来风险，所以不能像在线下压测的环境中那样，让资源使用率达到 100%，这也是要注意的一个要点。

因为前面我们也做了大量的工作，所以这里我们要把相应的记录表给清理一下。再次运行 JMeter 和 GoReplay 看看结果：

从这个数据可以看到，GoReplay 的 TPS 达到 1500 左右，JMeter 的 TPS 达到 150 左右，加在一起一共 1650 左右，比之前多了 300TPS。可见，数据量大了之后对性能的影响也是比较明显的，所以定时定量清理的策略是非常有必要的。

同时你也可以看到，在前面的截图中，有很多次都出现了分区使用率较高的情况，在这个过程中，由于我们的磁盘空间不太多（NFS 现在只有 100G），很快就用完了，所以我们还要不停去清理日志、监控数据清理等。但是为了不影响主线叙述逻辑，这些琐碎的工作我都没有体现在专栏中。不过你在做压测的时候，一定是要考虑到的。

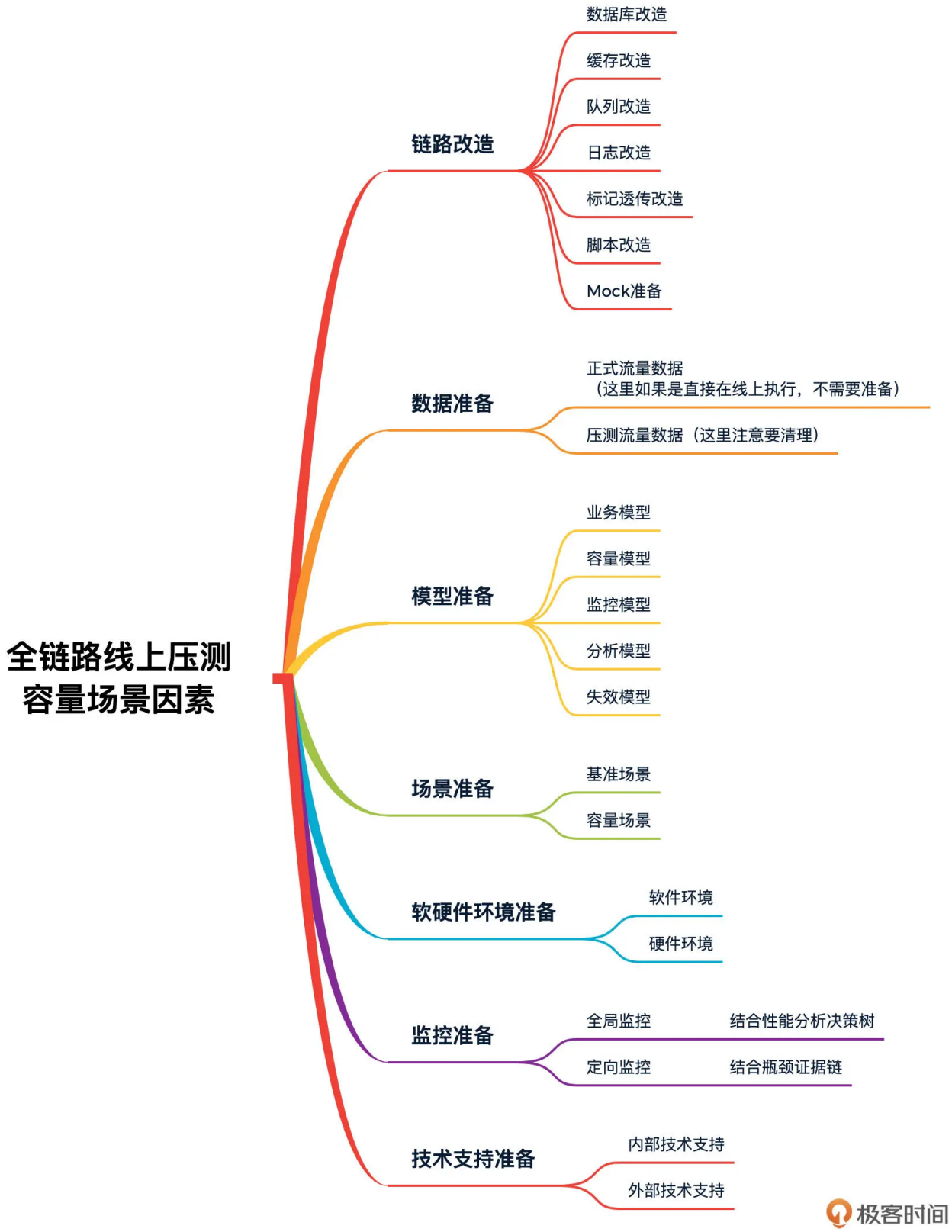
总结

好了，这节课就讲到这里。

刚才，我们同时启动了正常流量和压测流量，模拟了全链路压测场景应该有的样子。

我们全链路压测的目标呢，就是要把两种流量一起跑起来，并且让结果保持稳定。但是不要因为我们把这些场景都跑起来了，也没遇到啥问题，就觉得这个场景非常简单。其实，全链路压测的准备工作是非常多的，像我们之前讲的全链路的改造、数据的准备、两种压力流量脚本的准备、监控的准备等等都是要提前处理好的。我们是在做准备的时候就解决了疑难的技术点，才有了一个比较让人满意的结果。

那么，影响全链路线上压测容量场景的关键因素有哪些呢？我来画个思维导图说明一下：



其实在全链路线上压测的逻辑中，想同时跑起来两类场景，对企业的各方面支持都要求非常高。上图中的因素都是需要在具体落地时保证细致而精准的。

全链路线上压测，到这里，我觉得我已经把逻辑描述完整了。如果你一步步跟着学下来，就可以看到全链路压测的所有技术落地细节。全链路压测的实现并不是一个方案就可以描

述清楚的，只有通过这些技术的完整落地，才能实现真正的全链路线上压测。

课后题

学完这节课，请你思考两个问题：

1. 如果让你实现一个系统的全链路压测，你有明确的方向吗，可以分享一下。
2. 如果你的企业想要落地全链路压测，你觉得最大的阻碍在哪？

欢迎你在留言区与我交流讨论。当然了，你也可以把这节课分享给你身边的朋友，他们的一些想法或许会让你有更大的收获。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 预压测：如何基于小流量快速验证容量压测效果？

下一篇 32 | 稳定性场景：怎样搞定线上稳定性场景？

更多学习推荐

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。